

Esta clase va a ser

- grabada

**Clase 06.** FUNDAMENTOS DE LA CIENCIA DE DATOS

**NumPy**

# Objetivos de la clase



**Conocer** las estructuras de datos y su implementación en Python



**Entender** el uso básico del paquete NumPy

## MAPA DE CONCEPTOS



# Recall: estructuras de datos

✓ Anteriormente vimos las estructuras list, tuple, dict y set.

Tipo	Ejemplo	Definición
list	[1, 2, 3]	Lista ordenada
tuple	(1, 2, 3)	Lista ordenada inmutable
dict	{'a':1, 'b':2, 'c':3}	Diccionario: conjunto de pares clave:valor
set	{1, 2, 3}	Conjunto, a la manera de un conjunto matemático

# Numpy y ndarrays



# Probando estructuras en Python

Duración: 15 minutos



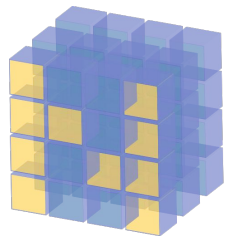
ACTIVIDAD EN CLASE

# Probando estructuras en Python

Deberán resolver dos problemas reales, utilizando las estructuras aprendidas de programación en Python en una notebook.



# Introducción a NumPy



NumPy

NUMerical PYthon

Potente estructura de  
datos

Implementa matrices y  
matrices  
multidimensionales

Estructuras que  
garantizan cálculos  
eficientes con matrices

NumPy es un proyecto de código abierto que tiene como objetivo permitir la computación numérica con Python. Fue creado en 2005, basándose en el trabajo inicial de las bibliotecas Numeric y Numarray.

NumPy siempre será un software 100% de código abierto, de uso gratuito para todos y publicado bajo los términos liberales de la licencia BSD modificada

*Equipo creador:*

<https://numpy.org/gallery/team.html>

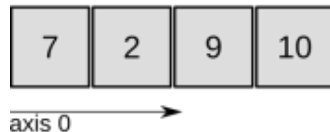
# El array como estructura de datos

- ✓ Extenderemos la aplicación de estos tipos de estructura de datos, agregando el tipo de dato **array**.
- ✓ Tanto array como list sirven para guardar conjuntos de datos **ordenados** en memoria.
- ✓ Mientras que el tipo de dato list puede guardar datos de diferentes tipos, el tipo de dato array guarda datos de un **único tipo**.
- ✓ Esto le permite ser **más eficiente**, especialmente al trabajar con conjuntos de datos grandes.

# El array como estructura de datos

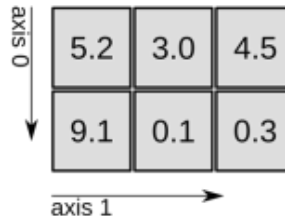
Los np.arrays pueden ser de diferentes dimensiones : 1D (vectores), 2D (matrices), 3D (tensores)

1D array



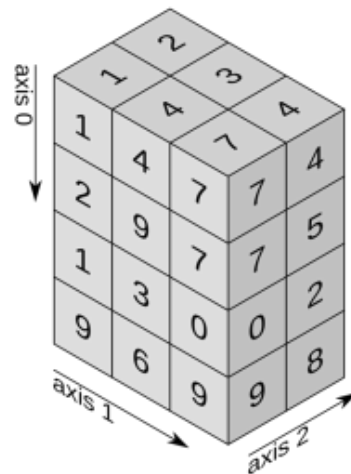
shape: (4,)

2D array



shape: (2, 3)

3D array



shape: (4, 3, 2)

# Creación de ndarrays

# Creación de ndarrays

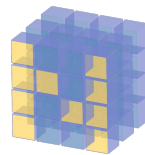
- ✓ La librería Numpy provee una forma particular de array llamada ndarray o Numpy Array.
- ✓ Recordar: los ndarrays, al ser un tipo de array, sólo pueden almacenar datos de un mismo tipo.

```
import numpy as np
```

```
Npa = np.array(range(10))  
Npa
```



```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```



# Veamos ejemplos

```
Np_cero = np.zeros(10)  
Np_cero
```

👉 `array([0., 0., 0., ..., 0.])`

```
Np_cero_int = np.zeros(10, dtype=int)  
Np_cero_int
```

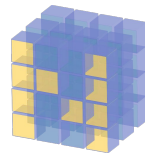
👉 `array([0, 0, 0, ..., 0])`

```
Np_uno = np.ones(10)  
Np_uno
```

👉 `array([1., 1., 1., ..., 1.])`

```
Np_relleno = np.full(10, 256)  
Np_relleno
```

👉 `array([256, 256, 256, ..., 256])`



# Veamos ejemplos

- ✓ Numpy provee objetos **rango**:

```
Np_rango = np.arange(10)
Np_rango
```



```
array([0, 1, 2, ..., 9])
```

- ✓ Ndarrays con **valores aleatorios** y de **dos dimensiones**:

```
Np_random_dimensiones = np.random.randint(10, size=(3, 4))
Np_random_dimensiones
```

```
array([[6, 5, 4, 7],
       [0, 1, 1, 2],
       [0, 0, 3, 8]])
```

# Tipos de datos y atributos de arrays



# Tipos de datos arrays

Tipos de datos en numpy	
i	integer -
b	boolean
u	unsigned integer
f	float
c	complex float
m	timedelta

Tipos de datos en numpy	
M	datetime
O	object
S	string
U	unicode string
V	fixed chunk of memory for other type ( void )

# Tipos de datos arrays

Tipos de datos en Python	
strings	para representar datos textuales
integer	para representar números enteros. e.g. -1, -2, -3
float	para representar números reales. e.g. 1.2, 42.42
boolean	para representar True o False.
complex	para representar números complejos. e.g. 1.0 + 2.0j, 1.5 + 2.5j

# Verificando el tipo de dato de un array

```
# Verificando el tipo de dato de array  
arr = np.array([1, 2, 3, 4])  
print(arr.dtype)
```



```
int64
```

```
arr = np.array(['apple', 'banana', 'cherry'])  
print(arr.dtype)
```



```
<U6
```

Creando arrays con formato específico

```
arr = np.array([1, 2, 3, 4], dtype='S')  
print(arr); print(arr.dtype)
```



```
[b'1' b'2' b'3' b'4']  
|S1
```

```
arr = np.array([1, 2, 3, 4], dtype='S')  
print(arr); print(arr.dtype)
```



```
[b'1' b'2' b'3' b'4']  
|S1
```

# Convertir el tipo de dato de un array

```
arr = np.array([1.1, 2.1, 3.1])  
newarr = arr.astype('i')  
print(newarr)  
print(newarr.dtype)
```



```
[1 2 3]  
int32
```

```
arr = np.array([1.1, 2.1, 3.1])  
newarr = arr.astype('i')  
print(newarr)  
print(newarr.dtype)
```



```
[ True False  True]  
bool
```

# Atribución de los arrays

# Ejemplos

```
Np_rango = np.arange(10)
Np_rango
```



```
array([0, 1, 2, ..., 9])
```

✓ Ndarrays con **valores aleatorios** y de **dos dimensiones**:

```
Np_random_dimensiones = np.random.randint(10, size=(3, 4))
Np_random_dimensiones
```

```
array([[6, 5, 4, 7],
       [0, 1, 1, 2],
       [0, 0, 3, 8]])
```

# Atributos de los Arrays

Inspeccionemos un poco nuestros Numpy arrays 🔍

Podemos acceder a distintas **propiedades** de los arreglos:



Dimensión:

```
Np_cero.ndim
```



```
1
```

```
Np_random_dimensiones.ndim
```



```
2
```



Forma:

```
Np_random_dimensiones.shape
```



```
(2, 3)
```



Tamaño:

```
Np_random_dimensiones.size
```



```
6
```

# Inspeccionemos nuestros Numpy arrays

Podemos acceder a distintas propiedades de los arreglos:

✓ Tipo de dato:

```
Np_cero.dtype
```



```
float64
```

```
Np_cero_int.dtype
```



```
int64
```

✓ Tamaño de elemento:

```
Np_random_dimensiones.itemsize
```



```
8
```

✓ Tamaño total:

```
Np_cero.nbytes
```



```
80
```

```
Np_cero_int.nbytes
```



# Resumen Tipos de Datos y Propiedades de Arrays

Sn	Cadena de texto (string) de n-caracteres
bool	Booleano (True o False). Se almacena como 1 bit
int	Entero (int32 o int64, dependiendo de la plataforma)
int8	Byte (-128 a 127)
int16	Entero (-32768 a 32767)
int32	Entero (-2.147.483.648 a 2.147.483.647)
int64	Entero (-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)
uint8	Entero sin signo (0 a 255)
uint16	Entero sin signo (0 a 65535)
uint32	Entero sin signo (0 a 4.294.967.295)
uint64	Entero sin signo (0 a 18.446.744.073.709.551.615)
float	Atajo para float64
float32	Decimal en precisión simple.
float64	Decimal en doble precisión.
complex	Atajo a complex128
complex64	Número complejo, parte entera e imaginaria con float32
complex128	Número complejo, parte entera e imaginaria con float64

Tipos de datos posibles en numpy array

# Resumen Tipos de Datos y Propiedades de Arrays

Propiedad	Descripción
<code>ndarray.shape</code>	Tupla con las dimensiones.
<code>ndarray.ndim</code>	Número de dimensiones.
<code>ndarray.size</code>	Número de elementos.
<code>ndarray.itemsize</code>	Tamaño de uno de los elementos en bytes.
<code>ndarray.nbytes</code>	Tamaño total ocupado por los elementos.
<code>ndarray.dtype</code>	Tipo de dato de los elementos.
<code>ndarray.real</code>	Parte real.
<code>ndarray.imag</code>	Parte imaginaria.

Propiedades de los numpy array

# Indexado y acceso

# Accediendo a elementos

## A tener en cuenta...

- Se accede a un elemento del array dando su **posición** en el array, mediante un índice ENTERO entre corchetes ( ' [] ' )

`nombre_array[posicion]`

- El **primer índice** es el 0 (como en C/C++).
- Si el **índice es mayor** que el número de elementos de array, **lanzará una excepción** (IndexError).

# Veamos cómo consultar los arreglos

- ✓ Al igual que las listas, los elementos del arreglo se acceden mediante su índice, comenzando desde 0.

```
rango = range(1,11)
Np_diez_numeros = np.array(rango)
Np_diez_numeros
```

👉 `array([ 1, 2, 3, ..., 10])`

- ✓ Primer elemento:

```
Np_diez_numeros[0]
```



1

- ✓ Quinto elemento:

```
Np_diez_numeros[4]
```



5

# Veamos cómo consultar los arreglos

- ✓ Podemos seleccionar elementos desde atrás para adelante mediante índices negativos, **comenzando desde -1**.

- ✓ Último elemento:

```
Np_diez_numeros[-1]
```



```
10
```

- ✓ Penúltimo elemento:

```
Np_diez_numeros[-2]
```



```
9
```

- ✓ Para acceder a un elemento de una matriz, indicar fila y columna:

```
Np_random dimensiones
```



```
array([[6, 5, 4, 7],  
       [0, 1, 1, 2],  
       [0, 0, 3, 8]])
```

```
Np_random dimensiones[2, 1]
```



```
0
```

## Tipo de selección

## Sintaxis

Un sólo elemento

`array[posicion]`

Varios elementos consecutivos

`array[inicio:fin]`

Elementos en orden cualesquiera

`array[[p1, p2,..., pn]]`

(Novedad respecto a PYTHON Core.)

donde `[p1, ...,pn]` es una lista o array.

## Recordatorio

Los índices pueden tomar valores negativos. Al igual que en las secuencias de PYTHON cuentan las posiciones desde el final del array.



# Accediendo a subarrays

# El array como estructura de datos

Podemos seleccionar una rebanada del arreglo de la siguiente manera:

```
Objeto[desde:hasta:tamaño_de_paso]
```

- ✓ El parámetro tamaño\_de\_paso permite, por ejemplo, seleccionar elementos de dos en dos
- ✓ Atención a estos detalles
  - El índice "desde" es inclusivo.
  - El índice "hasta" es exclusivo.

# Veamos algunos ejemplos

✓ Primeros cuatro:

```
Np_diez_numeros[:4]
```

```
[1 2 3 4]
```

✓ Desde el cuarto:

```
Np_diez_numeros[3:]
```

```
[ 4  5 ... 10]
```

✓ Desde el quinto al séptimo:

```
Np_diez_numeros[4:7]
```

```
[5 6 7]
```

✓ De dos en dos:

```
Np_diez_numeros[:,2]
```

```
[1 3 5 7 9]
```

✓ Desde atrás, de dos en dos:

```
Np_diez_numeros[::-2]
```

```
[10  8  6  4  2]
```

# Veamos algunos ejemplos

Para **arreglos multidimensionales**, especificar los índices de manera ordenada:

`Objeto[dimensión1, dimensión2,...]`

*Veamos algunos ejemplos...*

- Tercera fila, todas las columnas:
- Primeras dos filas, primeras dos columnas:
- Tercera fila, cuarta columna:

```
Np_random_dimensiones[2, ]
```

```
Np_random_dimensiones[:2, :2]
```

```
Np_random_dimensiones[2, 3]
```



# Break

¡10 minutos y volvemos!

# Operaciones básicas: reshape, concatenación, splitting

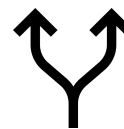
## Operaciones básicas



Reshape



Concatenación



Splitting

**Reshape**





# Para pensar

## Pensando en ajedrez

Si tuviésemos que rellenar una grilla de 8x8 con números desde 1 a 64, ¿Cómo lo haríamos?

# Reshape

Permite modificar la dimensión de un arreglo (siempre y cuando las dimensiones de salida están relacionadas con las de entrada)

¿Que patrón curioso observan?

Original  
(3, 4)

1	1	1	1
2	2	2	2
3	3	3	3

(6, 2)

1	1
1	1
2	2
2	2
3	3
3	3

(2, 6)

1	1	1	1	2	2
2	2	3	3	3	3

(4, 3)

1	1	1
1	2	2
2	2	3
3	3	3

(1, 12)

1	1	1	1	2	2	2	2	3	3	3	3
---	---	---	---	---	---	---	---	---	---	---	---

(12, 1)

1
1
1
1
2
2
2
2
3
3
3
3

# Reshape

Permite modificar la dimensión de un arreglo, retornando otro con distinta dimensión y forma pero manteniendo los mismos elementos.

```
np.arange(1,65)
```

👉 `array([ 1, 2, 3, ..., 64])`

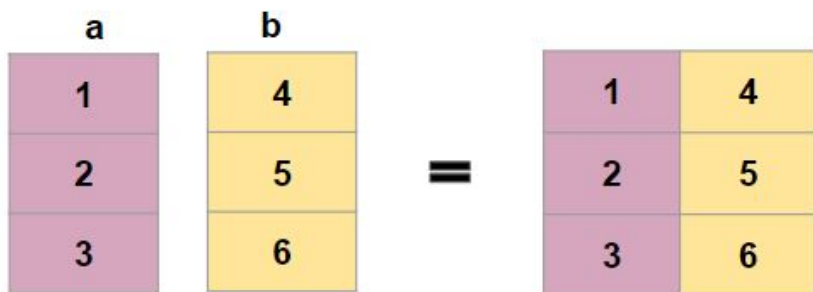
```
Ajedrez_64 = np.arange(1,65).reshape(8,8)  
Ajedrez_64
```

👉 `array([[ 1, 2, 3, 4, 5, 6, 7, 8],  
 [ 9, 10, 11, 12, 13, 14, 15, 16],  
 [17, 18, 19, 20, 21, 22, 23, 24],  
 [25, 26, 27, 28, 29, 30, 31, 32],  
 [33, 34, 35, 36, 37, 38, 39, 40],  
 [41, 42, 43, 44, 45, 46, 47, 48],  
 [49, 50, 51, 52, 53, 54, 55, 56],  
 [57, 58, 59, 60, 61, 62, 63, 64]])`

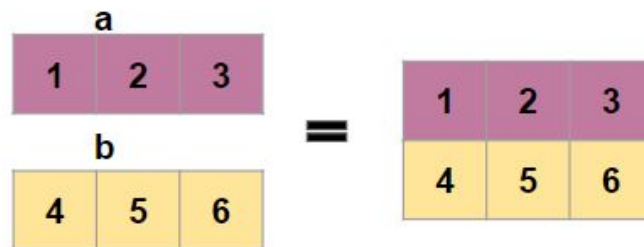
# Concatenación

# Concatenación

Permite modificar concatenar arrays siempre y cuando las dimensiones lo permitan.



axis= 1 concatena por columnas



axis= 0 concatena por filas

# Concatenación

Función de concatenación de arrays.

Función

```
numpy.concatenate((a1, a2, ...), axis=0)
```

con

- “(a1, a2, ...)” una secuencia de arrays. Su “shape” debe coincidir, a excepción de la dimensión dada por “axis”.
- “axis” es la dimensión donde se van a unir los arrays.

# Concatenación

Consiste en formar un nuevo arreglo a partir de “enganchar” o “apilar” otros.

- ✓ Python ofrece dos métodos:
  - Con la operación **concatenate**.
  - Con las operaciones **vstack** y **hstack**

```
Array_1 = np.random.randint(10, size=5)
Array_2 = np.random.randint(10, size=5)
Arrays_concatenados = np.concatenate([Array_1, Array_2])
```

👉 `array([0, 6, 5, 4, 3, 1, 8, 0, 0, 3])`

# Concatenación

- ✓ El método *vstack* **apila verticalmente**:

```
Array_extra = np.array([[10],[20]])  
Array_extra
```

👉 `array([[10],  
[20]])`

```
Array_apilados_v = np.vstack([Array_extra, Array_extra])  
Array_apilados_v
```

👉 `array([[10],  
[20],  
[10],  
[20]])`

- ✓ El método *hstack* **apila horizontalmente**:

```
Array_apilados_h = np.hstack([Array_extra, Array_extra])  
Array_apilados_h
```

👉 `array([[10, 10],  
[20, 20]])`



# Splitting

# Splitting

- ✓ Consiste en **desarmar** o **partir** los arreglos.
- ✓ Puede pensarse como la **operación inversa a la concatenación**

Arrays\_concatenados



```
array([0, 6, 5, 4, 3, 1, 8, 0, 0, 3])
```

```
Array_partido = np.split(Arrays_concatenados, [2])
```

```
Array_partido
```



```
[array([0, 6]), array([5, 4, 3, 1, 8, 0, 0, 3])]
```

Especificamos los **puntos de corte** con un arreglo. En este caso queremos un *único corte entre el segundo y tercer elemento*

# Splitting

✓ *Dos puntos de corte*

```
Array_partido_2 = np.split(Arrays_concatenados, [2, 8])  
Array_partido_2
```

```
[array([0, 6]), array([5, 4, 3, 1, 8, 0]), array([0, 3])]
```

✓ *Podemos desarmar el arreglo y guardarlo en variables distintas*



```
Parte_1, Parte_2, Parte_3 = Array_partido_2
```

Parte\_1

```
array([0, 6])
```

Parte\_2

```
array([5, 4, 3, 1, 8, 0])
```

Parte\_3

```
array([0, 3])
```

# Splitting

✓ *hsplit* realiza **cortes verticales**:

```
Ajedrez_partido_1 = np.hsplit(Ajedrez_64, [4])  
Ajedrez_partido_1
```



```
[array([[ 1,  2,  3,  4],  
       [ 9, 10, 11, 12],  
       [17, 18, 19, 20],  
       [25, 26, 27, 28],  
       [33, 34, 35, 36],  
       [41, 42, 43, 44],  
       [49, 50, 51, 52],  
       [57, 58, 59, 60]])],  
array([[ 5,  6,  7,  8],  
       [13, 14, 15, 16],  
       [21, 22, 23, 24],  
       [29, 30, 31, 32],  
       [37, 38, 39, 40],  
       [45, 46, 47, 48],  
       [53, 54, 55, 56],  
       [61, 62, 63, 64]])]
```

✓ *vsplit* realiza **cortes horizontales**:

```
Ajedrez_partido_2 = np.vsplit(Ajedrez_64, [4])  
Ajedrez_partido_2
```



```
[array([[ 1,  2,  3,  4,  5,  6,  7,  8],  
       [ 9, 10, 11, 12, 13, 14, 15, 16],  
       [17, 18, 19, 20, 21, 22, 23, 24],  
       [25, 26, 27, 28, 29, 30, 31, 32]])],  
array([[33, 34, 35, 36, 37, 38, 39, 40],  
       [41, 42, 43, 44, 45, 46, 47, 48],  
       [49, 50, 51, 52, 53, 54, 55, 56],  
       [57, 58, 59, 60, 61, 62, 63, 64]])]
```

# Agregaciones



# Cálculos sobre Numpy arrays

Como futuros Data Scientists, cotidianamente nos encontraremos con la tarea de efectuar **cálculos** a partir de arrays

Numpy está para darnos una mano en esto

# Calculando el promedio

Una *solución tradicional* al problema de calcular la media es la siguiente:

Si bien esta resolución es elegante y cumple con su tarea, Numpy nos provee de opciones más cómodas y eficientes 🚀

```
Array_aleatorio = np.random.randint(10, size=10)
print(Array_aleatorio)

suma = 0

for i in Array_aleatorio:
    suma += i

promedio = suma / np.size(Array_aleatorio)
```

# Agregaciones

✓ Suma:

```
Array_aleatorio.sum()
```

✓ Promedio:

```
Array_aleatorio.mean()
```

✓ Valor máximo:

```
Array_aleatorio.max()
```

✓ Mediana:

```
np.median(Array_aleatorio)
```

✓ Desvío estándar:

```
np.std(Array_aleatorio)
```

✓ Varianza:

```
np.var(Array_aleatorio)
```

Estas funciones están optimizadas para grandes volúmenes de datos y además nos ahorran **mucho código...** 😊



# Operaciones aritméticas

## Trigonómicas

sin(x)  
cos(x)  
tan(x)  
arcsin(x)  
arccos(x)  
arctan(x)  
hypot(x, y)  
degrees(x)  
radians(x)  
deg2rad(x)  
rad2deg(x)

## Redondeo

around(a[, decimals])  
round\_(a[, decimals,])  
rint(x)  
fix(x)  
floor(x)  
ceil(x)  
trunc(x)

## Hyperbólicas

sinh(x)  
cosh(x)  
tanh(x)  
acrsinh(x)  
arctan(x)

## Exp & Logs

exp(x)  
exp1m(x)  
exp2(x)  
log(x)  
log10(x)  
log2(x)  
log1p(x)

## Miscelánea

sqrt(x)  
power(x)  
fbas(x)  
sign(x)  
nan\_to\_num(x)

## Sum. & Prod. & Diff.

prod(a[, axis])  
sum(a[, axis])  
nansum(a[, axis])  
cumprod(a[, axis])  
cumsum(a[, axis])  
gradient(f, \*varargs)  
cross(a, b)

x e y son arrays. Las operaciones se realizan elemento a elemento. Las entradas a funciones trigonométricas que requieran de ángulos se dan en radianes.

# Operaciones estadísticas

Función	Descripción
<code>numpy.amin(a, axis=None)</code>	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis".
<code>numpy.amax(a, axis=None)</code>	Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis".
<code>numpy.nanmin(a, axis=None)</code>	Devuelve un array (o escalar) con el valor mínimo del array a lo largo del eje dado por "axis". Ignora los valores NaN.
<code>numpy.nanmax(a, axis=None)</code>	Devuelve un array (o escalar) con el valor máximo del array a lo largo del eje dado por "axis". Ignora los valores NaN.
<code>numpy.ptp(a, axis=None)</code>	Devuelve el rango de valores (máximo - mínimo) en el "axis" dado. El nombre de esta función viene del acrónimo "peak to peak".
<code>numpy.percentile(a, q, axis=None)</code>	Calcula y devuelve el percentil q-ésimo del array a en el eje "axis" especificado. <i>q</i> (escalar) en [0,100].


NOTA: Si `axis=None`, se trabaja sobre la versión transformada a 1D

Función	Descripción
<code>numpy.average(a, axis=None, weights=None)</code>	Devuelve un escalar o array con la media "pesada" del array <code>a</code> por los valores "weights" en el eje "axis" seleccionado. Los pesos pueden ser arrays 1-D, en cuyo caso ha de tener la misma longitud que <code>a</code> en el eje seleccionado. Si <code>weights=None</code> se asume el mismo peso (valor=1) para todos los elementos.
<code>numpy.mean(a, axis=None, dtype=None)</code>	Devuelve un escalar o array con la media aritmética del array sobre el "axis" dado. "dtype" establece el tipo de datos de entrada sobre el que promediar. El valor asignado por defecto es el del tipo del array.
<code>numpy.median(a, axis=None)</code>	Devuelve un escalar o array con la mediana del array para el eje seleccionado.
<code>numpy.std(a, axis=None, dtype=None, ..., ddof=0)</code>	Devuelve un escalar o array con la desviación estándar en el eje seleccionado. <code>ddof</code> es el acrónimo de <i>Delta Degrees of Freedom</i> . El denominador usado en los cálculos es $N - ddof$ , donde $N$ es el número de elementos.
<code>numpy.var(a, axis=None, dtype=None, ..., ddof=0)</code>	Devuelve un escalar o array con la varianza de los elementos del array en el eje seleccionado. Misma leyenda que <code>std</code> para el resto de parámetros.

# Operaciones vectorizadas

# Operaciones vectorizadas

¿Por qué son tan importantes?

- ✓ Incluso las operaciones más sencillas pueden resultar muy lentas si las llevamos a cabo elemento a elemento.
- ✓ Las computadoras son especialmente buenas para realizar cálculos en **paralelo** 
- ✓ Las **operaciones vectorizadas** o **funciones universales (ufuncs)** nos permiten operar entre arreglos de la manera más rápida posible.

# Operemos arreglos, pero de manera eficiente

Recordemos los arreglos de prueba:

Array\_1



```
array([7, 0, 6, 7, 5])
```

Array\_2



```
array([3, 7, 9, 9, 0])
```

✓ Sumas vectorizadas:

Array\_1 + 5



```
array([12, 5, 11, 12, 10])
```

Array\_1 + Array\_2



```
array([10, 7, 15, 16, 5])
```

np.add(Array\_1, Array\_2)

¡Ambas formas son equivalentes!

# Producto Vectorial

- ✓ El **producto vectorial** sobre arreglos *unidimensionales* se calcula sumando los resultados de multiplicar los elementos que tienen la misma posición.

```
np.matmul(Array_1, Array_2)
```



138

- ✓ En Numpy, la versión vectorizada se implementa en el método `np.matmul`

## CLASE N°6

# Glosario

**Numpy:** librería de Python que nos permite trabajar con matrices y vectores de forma sencilla y potente

**Array:** estructura fundamental en Numpy que solo permite un solo tipo de dato haciéndolo eficiente para operaciones de alta complejidad, pueden ser de 1D (vectores), 2D (matrices) o 3D (tensores)

**Atributos de arrays:** son las propiedades de los arrays creados, podemos extraer propiedades como: dimensión (.ndim), forma (.shape), tamaño (.size) entre otros

**Indexación:** forma de extraer elementos de un objeto en Python. Importante recordar que el primer índice es el 0 de izquierda a derecha y de derecha a izquierda es -1.

**Reshape:** modificar la forma de un array siempre y cuando las dimensiones de entrada y salida sean compatibles

**Concatenación:** apilamiento de arrays siempre y cuando las dimensiones sean compatibles

**Splitting:** desarmado de un array (operación inversa de la concatenación)

**Agregaciones:** todas aquellas funciones preestablecidas que nos permiten calcular medidas de tendencia central (e.g media, mediana) o dispersión (e.g. varianza ,desviación estándar) de manera eficiente



# Resumen de la clase hoy

- ✓ Introducción a NumPy y ndarrays, acceso e indexado.
- ✓ Operaciones básicas con ndarrays.
- ✓ Agregaciones
- ✓ Operaciones vectorizada

**Muchas gracias.**



**¡Opina y valora esta  
clase!**

**#DemocratizandoLaEducación**